ECE 530: Cloud Computing

Homework #3: Linux Containers

Amber Disher – 101839171 – adisher1@unm.edu

Marshall Hundemer – 101736010 – mhundemer@unm.edu

David Kirby – 101652098 – davidkirby@unm.edu

Spring 2021

School of Engineering

Contents

List of Figures

2 Abstract

Docker is a platform to easily maintain highly configurable instances. It can be set up and ran in milliseconds, and can create globally accessible services. For homework #3 we were tasked to create a Dockerfile that can build images automatically, then to deploy a distributed database based on Linux containers. Our deployment must contain at least two containers and therefore at least two database instances. The instances must be connected to each other and contain part of the data.

3 Introduction

For our deployment we chose MongoDB, a document-oriented NoSQL database. To create our images we used the Dockerfile shown in the Appendix. We quickly learned that Dockerfiles are limited in their build capabilities, notably with creating networks and creating multiple images at once. These issues can be solved using docker-compose, but that is beyond the scope of this assignment.

4 Deployment

In order to use Docker, we needed to install the necessary software. For reference, our host machine is running macOS Big Sur on an Apple silicon-based MacBook Air. We installed Docker using Homebrew, a package manager for macOS, and needed to run the latest Docker version in order to work on ARM processors. One of the most important features of Docker allows us to build on ARM, but then deploy on other architectures as well.

Install Docker via Homebrew

```
    brew install docker
```

Docker verification

```
# Verify Docker version

    docker -v

# Check Docker images

    docker images
```

Pulls latest version of MongoDB from Docker Hub

```
    docker pull mongo
```

The output of these commands is shown in Figure 1.

```
-zsh
(base) ~: docker -v
Docker version 20.10.6, build 370c289
(base) ~: docker images
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
<none>        <none>    9ffa3c88c36b   2 hours ago     454MB
mongo         latest    cc8582bb045a   13 days ago     424MB
(base) ~: docker pull mongo
Using default tag: latest
latest: Pulling from library/mongo
673aeee5c81c: Already exists
018b2790219d: Already exists
509c77ce92ad: Already exists
0cd458c20dd1: Already exists
fb1c02642395: Already exists
12d074f6952b: Already exists
811aaa41efc3: Already exists
de5b8f707197: Already exists
a5d9ce6f018d: Already exists
aad8298286a9: Already exists
f8716c73dec1: Already exists
27facd4ec39b: Pull complete
Digest: sha256:0145cc5df5e657154b83fa04b90e48f6f89b608850d0906addfc0d1681a66a1e
Status: Downloaded newer image for mongo:latest
docker.io/library/mongo:latest
(base) ~:
```

Figure 1: Verification of Docker Installation.

Once Docker was installed we needed to set up a network to which our databases could join. Docker has default networks installed:

- host – For standalone containers, removes network isolation between the container and the Docker host, and uses the host's networking directly.
- bridge – The default network driver. Bridge networks are usually used when your applications run in standalone containers that need to communicate.

The default bridge network allow us to configure the network, but all the containers use the same settings, such as MTU and iptables rules. In addition, configuring the default bridge network happens outside of Docker itself, and requires a restart of Docker. Creating our own bridge network, created and configured using `docker network create`, allows different groups of applications to have different network requirements, allows us to configure each user-defined bridge separately. Containers connected to the same user-defined bridge network effectively expose all ports to each other.

List all networks created in Docker

```
docker network ls
```

Create our user-defined network

```
docker network create mongo-net
```

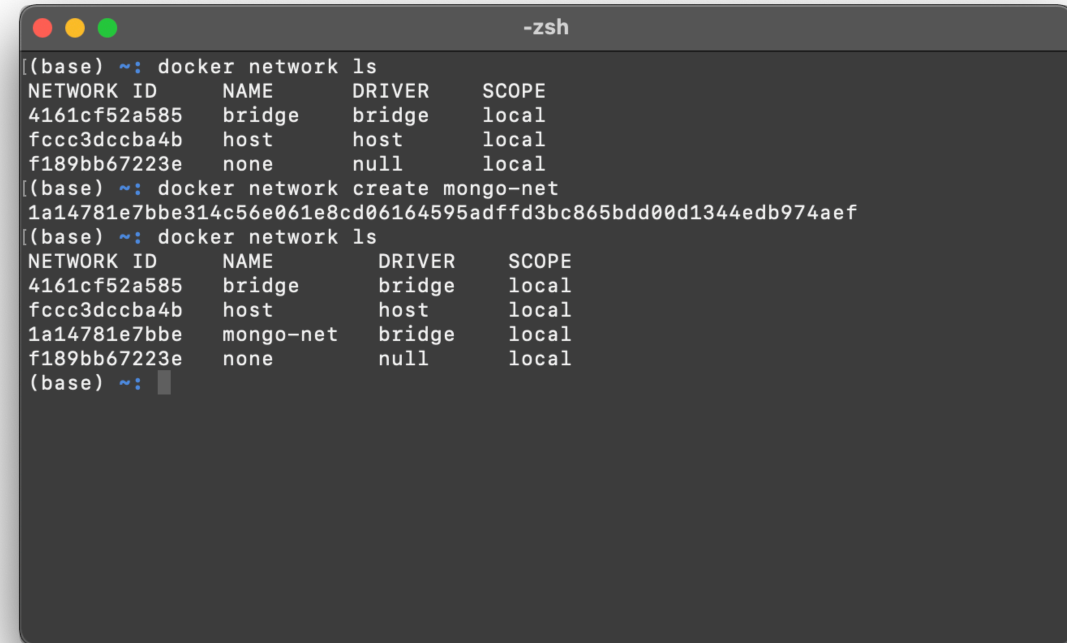

Figure 2: Creation of User-defined Network.

Now that our network was created (see Figure 2), we began creating our containers from the Mongo image and connecting them altogether.

Create first MongoDB image

```
docker run -p 30001:27017 --name mongo1 --net mongo-net mongo \
mongod --replSet mongo-set
```

- `docker run` – Start a container from an image
- `-p 30001:27017` – Expose port 27017 in our container, as port 30001 on the localhost
- `--name mongo1` – Name this container "mongo1"
- `--net mongo-net` – Add this container to the "mongo-net" network.
- `mongo` – the name of the image we are using to spawn this container
- `mongod --replSet mongo-set` – Run mongod while adding this mongod instance to the replica set named "mongo-set"

We can see in Figure 3 that we are running Ubuntu 18.04.5 LTS and MongoDB 4.4.5 on aarch64 (ARM) architecture.

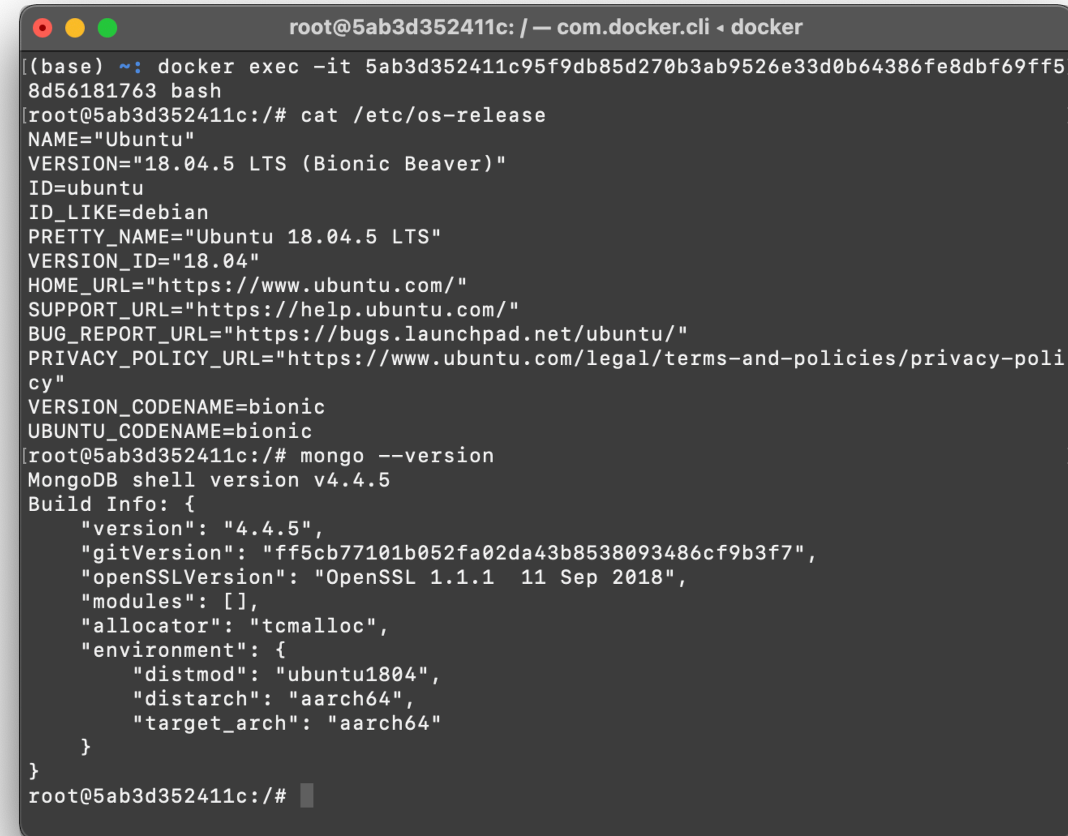

Figure 3: Ubuntu and MongoDB Version Checks.

We then created the secondary images, each of which needed to be run in a separate terminal tab, and finally after all database containers were created, we turned them into a replica set.

Create secondary MongoDB images

```
docker run -p 30002:27017 --name mongo2 --net mongo-net mongo \
mongod --replSet mongo-set

docker run -p 30003:27017 --name mongo3 --net mongo-net mongo \
mongod --replSet mongo-set
```

We started our Docker container directly into our MongoDB and began configuring the connections.

Connect to `mongo1` and configure it to be the primary

```
docker exec -it mongo1 mongo

> db = (new Mongo('localhost:27017')).getDB('test')

> config = {
    "_id" : "mongo-set",
    "members" : [
    {
        "_id" : 0,
        "host" : "mongo1:27017"
    },
    {
        "_id" : 1,
        "host" : "mongo2:27017"
    },
    {
        "_id" : 2,
        "host" : "mongo3:27017"
    }
    ]
  }
```

The results of the above commands are shown in Figure 4. Then we initiated the configuration file we just created and received confirmation by noticing the change in command prompt.

Initiate our replica set using our just created config file

```
> rs.initiate(config)

mongo-set:PRIMARY> # Confirms that we are on PRIMARY DB
```

We then wrote data to our primary DB to verify it (see Figure 5), writing a document "ECE530". It is important to note that data can only be written to the primary DB, but, as we will show later, can be read from any of the databases. If for some reason the primary database goes down, one of the others will become primary and allow for robustness.

Write data to `mongo1` – our primary DB and then read it

```
> db.mycollection.insert({name : 'ECE530'})

> db.mycollection.find()
```

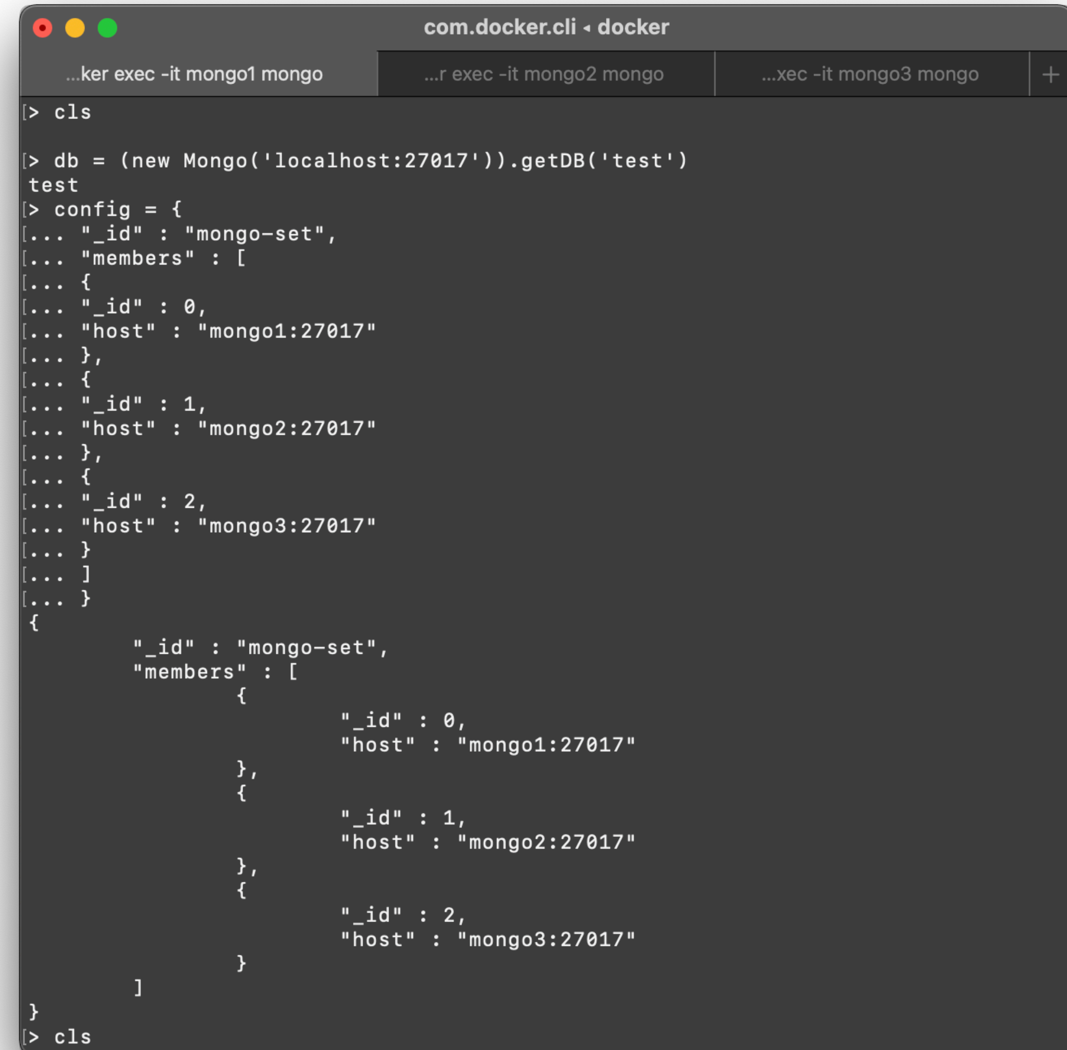

Figure 4: Database Connection and Configuration.

We then connect to each of our secondary databases and test to see if our document gets replicated there as well.

Test that data is being replicated to `mongo2` and `mongo3` respectively

```
> db2 = (new Mongo('mongo2:27017')).getDB('test')

> db2.setSecondaryOk()

> db2.mycollection.find()


> db3 = (new Mongo('mongo3:27017')).getDB('test')

> db3.setSecondaryOk()

> db3.mycollection.find()
```

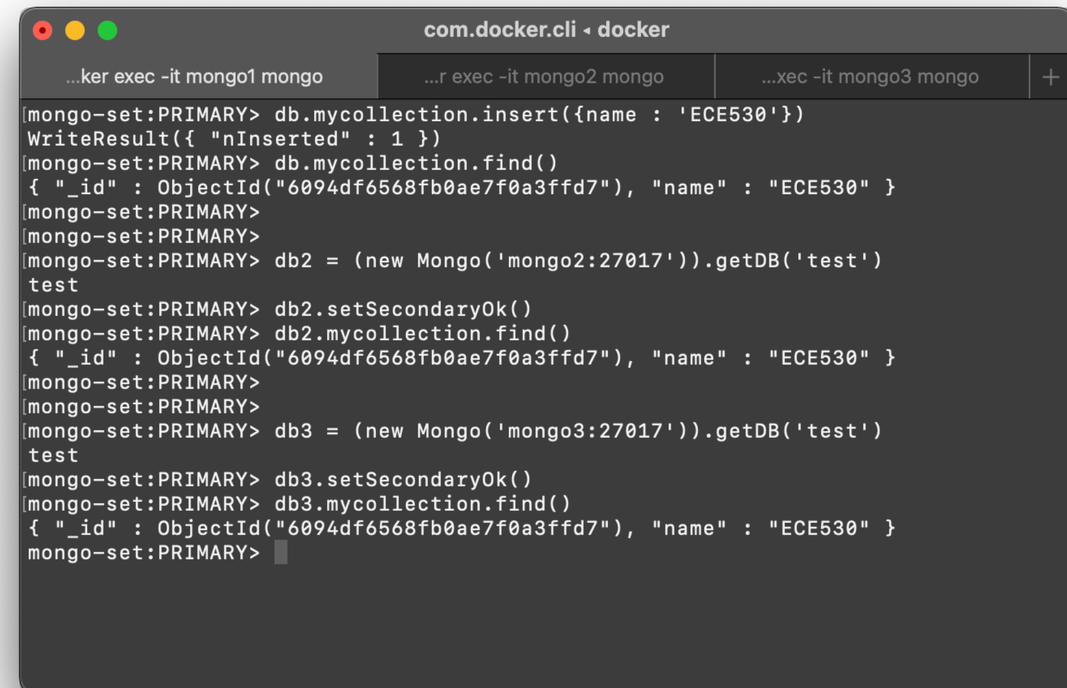

Figure 5: Database Write, Read, and Replication.

5 Conclusion

In conclusion, we were successful in creating a Dockerfile to create images of MongoDB, although we found Dockerfiles limited. We set up a user-defined network for our databases, created three MongoDB databases from our Docker image, and designed those databases to be a replica set. We tested writing data to our primary database and confirmed that the data is indeed distributed. For future work, we would like to explore docker-compose to attempt automating this entire process.

6 Extra

While Docker makes configuring and accessing containers extremely easy using the command line interface, it is also very helpful to view our images and containers using a graphical user interface. For example, we show our created replica set in both the Docker Desktop (Figure 6) and in Visual Studio Code, which has built-in Docker support. Using VSCode, we can even view files within Ubuntu image (Figure 7), view all of our containers, images, networks, volumes, and much more at a glance, and even start, stop, and remove items.

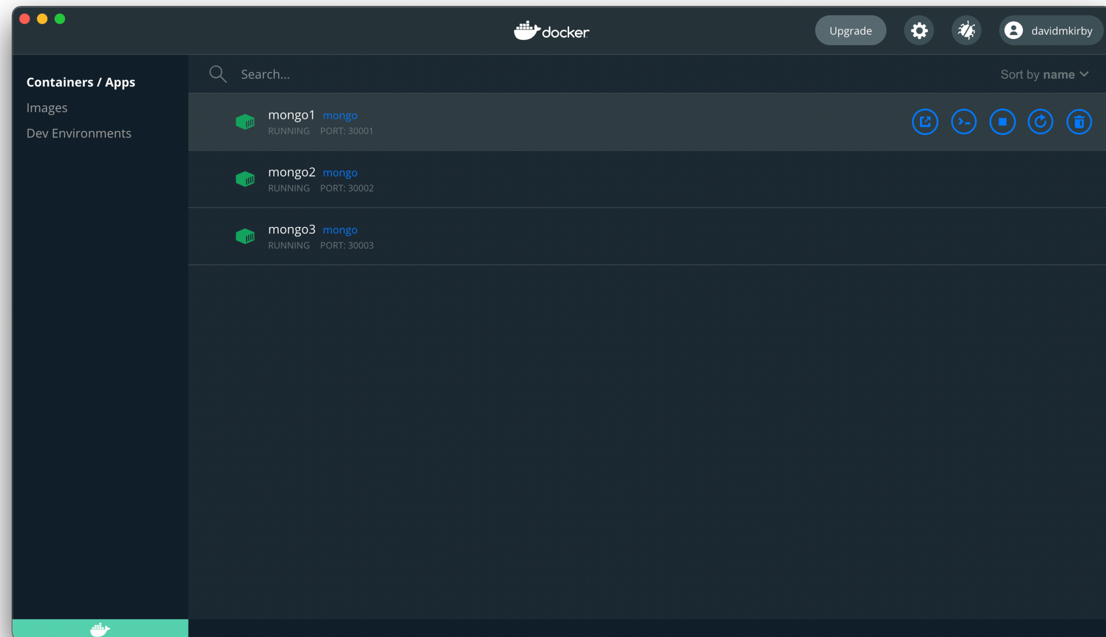

Figure 6: Docker Desktop.

7 Appendix

Dockerfile

```
# Pulls latest version of MongoDB from Docker Hub

    FROM mongo
    LABEL maintainer=davidkirby@unm.edu

# Automatically update Ubuntu

    RUN apt-get update && apt-get install -y

# Open Port for MongoDB to connect to host

    EXPOSE 27017
```

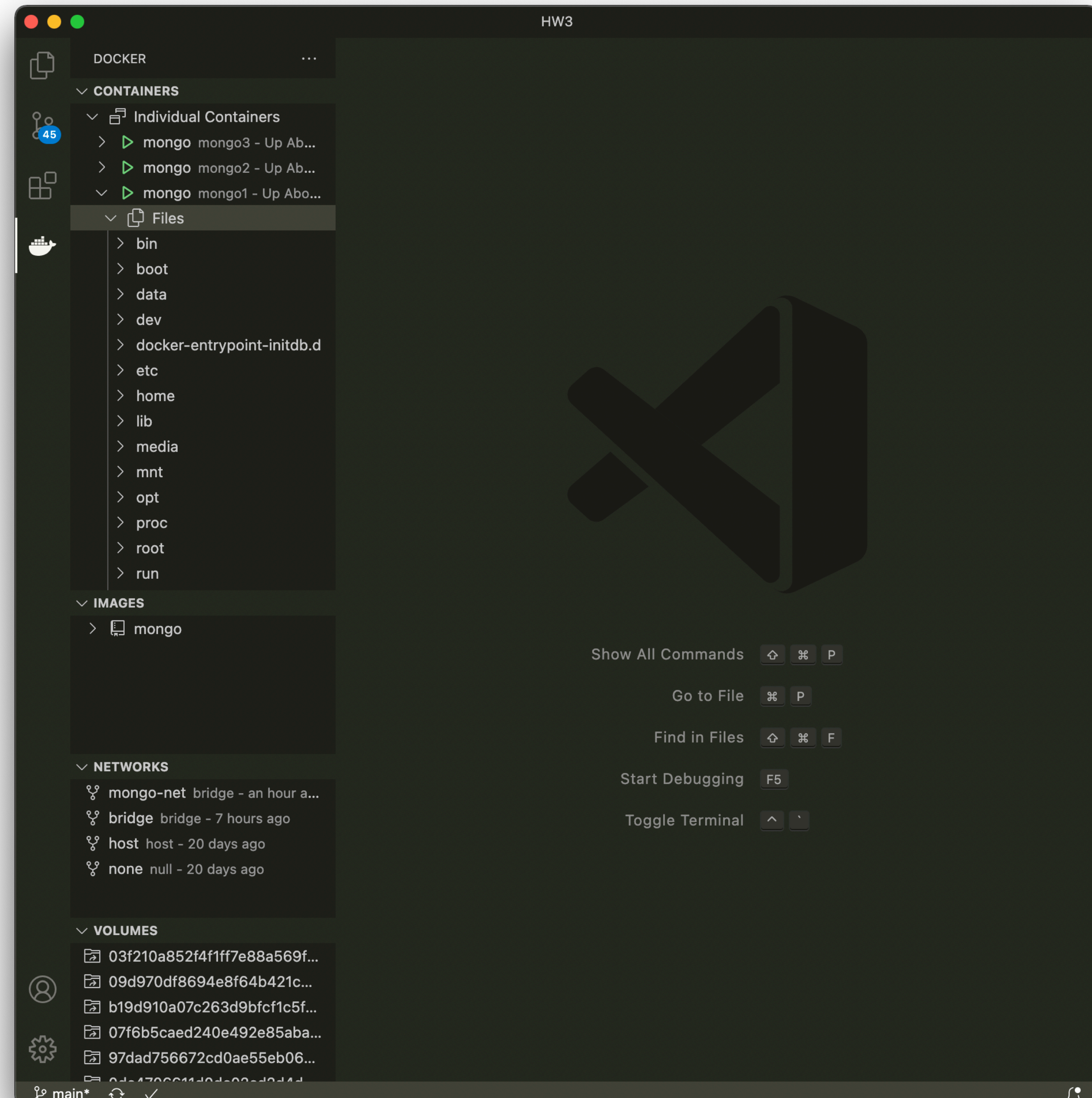

Figure 7: Microsoft Visual Studio Code Docker Support.